Intel® Atom™ Processor E3800 Windows 7 IO Driver Software Developer's Manual

December 2013

Contents

CHAPTER 1
About this Manual

1.1 Operating System Covered in This Manual

This manual set includes information pertaining to the following set of Operating system

- Windows 7 Ultimate 32 bit SP1
- Windows 7 Ultimate 64 bit SP1
- Windows Embedded Standard 32 bit SP1
- Windows Embedded Standard 64 bit SP1

The IO drivers are dependent on the Operating System (OS) driver installation.

Note: Minor update to GPIO, I2C and SPI driver on structure definition in public driver header file from beta driver to gold driver. Recompile your applications with the latest public driver header.

CHAPTER 2
General Purpose Input Output (GPIO) Driver

This section provides the programming details and interfaces exposed by the General Purpose Input Output (GPIO) driver for Windows. The current implementation of the driver exposes the interfaces through Input / Output Controls (IOCTLs), which can be called from the application (user mode) using the Win32 API DeviceIoControl (Refer to the MSDN documentation for more details on this API). The following sections provide information about the IOCTLs and how to use them to configure the GPIO hardware.

2.1 Driver Features

The GPIO Driver supports:

- Setting of different function for GPIO hardware
- Writing data to GPIO hardware
- Reading data from GPIO hardware
- Setting the direction of GPIO hardware
- Querying the function of GPIO hardware

2.2 Interface Details

Table 1 lists IOCTLs supported by the driver.

No	IOCTL	Remarks
1	IOCTL_GPIO_READ	Read the data of selected pin of given GPIO controller
2	IOCTL_GPIO_WRITE	Write the data of selected pin of given GPIO controller
3	IOCTL_GPIO_DIRECTION	Set the direction of the selected pin of given GPIO controller
4	IOCTL_GPIO_MUX	Set the function of the selected pin of given GPIO port
5	IOCTL_GPIO_QUERY	Query the function of the selected pin of given GPIO port

Table 1. Supported IOCTLs

2.3 IOCTL Usage Details

This section assumes a single client model where there is a single application-level program configuring the GPIO interface and initiating I/O operations. The following files contain the details of the IOCTLs and data structures used:

- public.h – contains IOCTL definitions, data structures and other variables used by the IOCTLs

2.3.1 IOCTL_GPIO_MUX

This IOCTL is called to set the function mode of the selected pin of given GPIO controller. The prerequisite for this is that the device must be installed and opened using the Win32 API CreateFile.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.data = function;
DeviceIoControl(hHandle,
            IOCTL_GPIO_DIRECTION,
            & GPIO_PIN_PARAMETERS,
            sizeof(GPIO_PIN_PARAMETERS),
            NULL,
            0,
            &dwSize,
            NULL);
```

2.3.2 IOCTL_GPIO_DIRECTION

This IOCTL is called to set the direction of the selected pin of given GPIO controller. The prerequisite for this is that the device must be installed and opened using the Win32 API CreateFile and the pin is set to GPIO function mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.ConnectMode = direction;
DeviceIoControl(hHandle,
            IOCTL_GPIO_DIRECTION,
            & GPIO_PIN_PARAMETERS,
            sizeof(GPIO_PIN_PARAMETERS),
            NULL,
            0,
            &dwSize,
            NULL);
```

2.3.3 IOCTL_GPIO_READ

Read the data of selected pin of given GPIO controller. The prerequisite for this is that the device must be installed and opened using the Win32 API CreateFile.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
DeviceIoControl(hHandle,
      IOCTL_GPIO_READ,
      &parameter,
      sizeof(GPIO_PIN_PARAMETERS),
      &parameter,
      sizeof(GPIO_PIN_PARAMETERS),
      &dwSize,
      NULL);
```

2.3.4 IOCTL_GPIO_WRITE

The write operation writes to the selected pin of the GPIO controller. The prerequisite for this is that the device must be installed and opened using the Win32 API CreateFile and the pin direction is set to output.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.data = ConnectModeOutput;
DeviceIoControl(hHandle,
            IOCTL_GPIO_DIRECTION,
            &parameter,
            sizeof(GPIO_PIN_PARAMETERS),
            &parameter,
            sizeof(GPIO_PIN_PARAMETERS),
            &dwSize,
            NULL);

parameter.pin = pin;
DeviceIoControl(hHandle,
      IOCTL_GPIO_WRITE,
      &parameter,
      sizeof(GPIO_PIN_PARAMETERS),
      &parameter,
      sizeof(GPIO_PIN_PARAMETERS),
      &dwSize,
      NULL);
```

2.3.5 IOCTL_GPIO_QUERY

This IOCTL is called to query the function mode of the selected pin of given GPIO controller. The prerequisite for this is that the device must be installed and opened using the Win32 API CreateFile

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
DeviceIoControl(hHandle,
      IOCTL_GPIO_QUERY,
      &parameter,
      sizeof(GPIO_PIN_PARAMETERS),
      &parameter,
      sizeof(GPIO_PIN_PARAMETERS),
      &dwSize,
      NULL);
```

2.4 Structures, Enumeration and Macros

This section provides the details on the structures, enumerations and macros used by interfaces exposed by the GPIO driver. All the structures, enumerations and macros used by the interfaces are defined in public.h.

2.4.1 Structures

GPIO Pin Parameters

This structure is a used for preserving information related to the GPIO request.

Name	Description
ULONG pin	Select the pin number
union { ULONG data; GPIO_CONNECT_IO_PINS_MODE ConnectMode; } u;	Data in the case of read return the read pin value, Data in the case of write is the data to be written to the pin, Data in the case of mux is the function to be set to the pin, Data in the case of query return the function of pin. ConnectMode in the case of direction set the direction of the pin.

2.4.2 Enumeration

GPIO_CONNECT_IO_PINS_MODE

This enum is used for preserving information related to the direction.

Name	Description
CONNECT_MODE_INPUT	Set direction as input
CONNECT_MODE_OUTPUT	Set direction as output

2.4.3 Macros

Currently there are no macros defined for the GPIO driver.

2.5 Error Handling

Since the IOCTL command is implemented using the Windows API, the return value of the call is dependent on and defined by the OS. On Windows, the return value is a non-zero value. If the error is detected within or outside the driver, an appropriate system defined value is returned by the driver.

2.6 Inter-IOCTL Dependencies

There are no inter-IOCTL dependencies for GPIO driver. Once the driver is loaded successfully, the IOCTLs stated above can be used in any order.

2.7 Programming Guide for GPIO Driver

This section describes the basic procedure for using the GPIO driver from a user mode application. All operations are through the IOCTLs exposed by the GPIO driver. Refer to Section 4.3 for details on the IOCTLs. The steps involved in accessing the GPIO driver from the user mode application are described below:

- Open the device
- Initialize and configure the driver with desired settings through the interfaces exposed.
- Perform read/write operations.
- Close the device.

2.7.1 Opening the Device

The GPIO driver is opened using the Win32 CreateFile API.

Using GUID Interface Exposed by the driver

A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, and then enable an instance of the class for each device object to which user-mode I/O requests might be sent. The GPIO driver registers the following interface.

No	Interface Name
1	GUID_DEVINTERFACE_GPIO

This is defined in public.h. Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code can use SetupDiXxx functions to find out about registered, enabled device interfaces.

Please refer the following site to get the details about SetupDiXxx functions.
http://msdn.microsoft.com/en-us/library/ff549791.aspx

There are three GPIO controllers in the system, you should first determine which GPIO controller you want to open. By checking the path name returned by call SetupDiGetDeviceInterfaceDetail, you can know the controller type. If the device path returned start with “\\?\acpi#int33b2#1”, it means this controller is GPIO SCORE, if the device path returned start with “\\?\\acpi#int33b2#2”, it means this controller is GPIO NCORE, if the device path returned start with “\\?\\acpi#int33b2#3”, it means this controller is GPIO SUS.

2.7.2 Driver Configuration

The following IOCTLS are used to initialize, configure and query the settings for the GPIO driver:

- IOCTL_GPIO_DIRECTION
- IOCTL_GPIO_MUX
- IOCTL_GPIO_QUERY

DeviceIoControl Win32 API is used for sending information to the GPIO driver.

Direction Operation

This IOCTL used to set the pin direction when pin is in GPIO function mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.ConnectMode = direction
DeviceIoControl(hHandle,
           IOCTL_GPIO_DIRECTION,
           &parameter,
           sizeof(GPIO_PIN_PARAMETERS),
           &parameter,
           sizeof(GPIO_PIN_PARAMETERS),
           &dwSize,
           NULL);
```

The parameter.u.ConnectMode is used to set the pin direction.

Mux Operation

This IOCTL used to set pin to select function mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.data = function;
DeviceIoControl(hHandle,
           IOCTL_GPIO_MUX,
           &parameter,
           sizeof(GPIO_PIN_PARAMETERS),
           &parameter,
           sizeof(GPIO_PIN_PARAMETERS),
           &dwSize,
           NULL);
```

The parameter.u.data is used to set the pin function.

Query Operation

This IOCTL used to query the pin function mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
DeviceIoControl(hHandle,
           IOCTL_GPIO_QUERY,
           &parameter,
           sizeof(GPIO_PIN_PARAMETERS),
           &parameter,
           sizeof(GPIO_PIN_PARAMETERS),
           &dwSize,
           NULL);
```

The parameter.u.data is used to save the returned pin function value.

2.7.3 Read and Write Operation

IOCTL_GPIO_READ and IOCTL_GPIO_WRITE are used for read and write operations respectively.

Read Operation

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
DeviceIoControl(hHandle,
            IOCTL_GPIO_READ,
            &parameter,
            sizeof(GPIO_PIN_PARAMETERS),
            &parameter,
            sizeof(GPIO_PIN_PARAMETERS),
            &dwSize,
            NULL);
```

The parameter.u.data is used to save the return pin value.

Write Operation

To write a value to a pin, the pin must first set to output mode.

```
GPIO_PIN_PARAMETERS parameter;
parameter.pin = pin;
parameter.u.ConnectMode = ConnectModeOutput;
DeviceIoControl(hHandle,
            IOCTL_GPIO_DIRECTION,
            &parameter,
            sizeof(GPIO_PIN_PARAMETERS),
            &parameter,
            sizeof(GPIO_PIN_PARAMETERS),
            &dwSize,
            NULL);

parameter.pin = pin;
parameter.u.data = value;
DeviceIoControl(hHandle,
            IOCTL_GPIO_WRITE,
            &parameter,
            sizeof(GPIO_PIN_PARAMETERS),
            &parameter,
            sizeof(GPIO_PIN_PARAMETERS),
            &dwSize,
            NULL);
```

The parameter.u.data is used to set the value write to the pin.

2.7.4 Close the Device

Once all the operations related to the GPIO driver are finished, the device handle must free the application by calling the Win32 API CloseHandle.

```
CloseHandle(hHandle);
```

CHAPTER 3
Inter Integrated Circuit (I2C) Driver

This section describes the programming details of the Inter Integrated Circuit (I2C) driver for Windows 7. This includes the information about the interfaces exposed by the driver and how to use the interfaces to drive the I2C hardware through Input/Output Controls (IOCTLs), which can be called from the application (user mode) using the Win32 API DeviceIoControl. Refer to the MSDN documentation for more details on this API.

I2C (Inter-Integrated Circuit) is a multi-master serial computer bus that is used to attach low-speed peripherals to a motherboard or embedded system. I2C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock (SCL), pulled up with resistors.

3.1 Driver Features

The I2C Driver supports:

- Setting different configurations for I2C hardware.
- Master device only.
- Setting I2C slave device address.
- Mode Select – fast mode (400 kbps) or standard mode (100 kbps) only.
- I2C Bus Master byte/multi-byte read transactions.
- I2C Bus Master byte/multi-byte write transactions.

3.2 Interface Details

No	IOCTL	Remark
1	IOCTL_I2C_EXECUTE_WRITE	Configure slave address, address mode and speed, and then write data to the assigned slave device.
2	IOCTL_I2C_EXECUTE_READ	Configure slave address, address mode and speed, and then read data from the assigned slave device.
3	IOCTL_I2C_EXECUTE_SEQUENCE	Process a serial of Reads/Writes. Each one can have its own configuration. NOTE: Only ONE STOP bit will be produced after all items of one sequence done. So two independent serials should not be combined into one sequence, if each of them must produce STOP bit respectively after complete.

3.3 Structures and Macros

3.3.1 Structures

enum I2C_BUS_SPEED

This enum defines the I2C transmission speeds.

enum I2C_ADDRESS_MODE

This enum defines the address modes for slave device.

struct I2C_SINGLE_TRANSMISSION

This structure contains transmission data and I2C bus configuration.

struct I2C_SEQUENCE_TRANSMISSION

This structure contains one transmission data of one item in a sequence and related I2C bus configuration.

3.3.2 Macros

I2C_SEQUENCE_TRANSMISSION_ENTRY

This macro helps initialize a sequence structure, which can contain more than one read/write item.

I2C_SEQUENCE_ITEM_INIT

This macro initializes related I2C configuration and data buffer pointer of one item in a sequence transmission.

3.4 Error Handling

Since the IOCTL command is implemented using the Windows API, the return value of the call is dependent on and defined by the OS. On Windows, the return value is a non-zero value. If the error is detected within or outside the driver, an appropriate system defined value will be returned by the driver.

3.5 Programming Guide

This section explains the basic procedure to use the I2C driver from a user application mode. All operations are performed through the IOCTLs that are exposed by the I2C driver.

3.5.1 Open Device

The I2C driver is opened using the Win32 CreateFile API. To get the device name, use GUID interface exposed by the driver: **I2C_LPSS_INTERFACE_GUID**, defined in public.h.

A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, and then enable an instance of the class for each device object to which user-mode I/O requests might be sent.

Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code can use SetupDiXxx functions to find out about registered, enabled device interfaces. Please refer the following site to get the details about SetupDiXxx functions.

http://msdn.microsoft.com/en-us/library/dd406734.aspx

Since there are more than one I2C controller in BYT-I platform, and they share the same GUID, when user-mode applications open I2C device using SetupDiXxx, they will get a device name list of all I2C controller interfaces. At this time, they should also compare the hardware ID they need to each item of that list, so as to be able open the correct controller they need.

3.5.2 Read, Write, and Sequence Operation

IOCTL_I2C_EXECUTE_READ, IOCTL_I2C_EXECUTE_WRITE and IOCTL_I2C_EXECUTE_SEQUENCE are used for read, write, and sequence operation respectively.

(Maximum single transfer size is 64k, but this value may be updated in further, check the platform user guide for latest value)

Transmission block initialization

Before doing transmission, a transmission structure variable must be defined in advance.

For example:

```
I2C_SINGLE_TRANSMISSION transmission;
```

The Application should use asynchronous method of IOCTL to do read/write operation. So before using DeviceIoControl, structure Overlapped must be initialized first. Please refer to the following link to get detailed information:

http://msdn.microsoft.com/en-us/library/windows/desktop/ms686358%28v=vs.85%29.aspx

IOCTL_I2C_EXECUTE_READ Code Example

```
#include “I2Cpublic.h”

I2C_SINGLE_TRANSMISSION readTransmission;
UCHAR readBuf[BUF_SIZE] = {};
UINT16 slaveAdr         = 0x1C;


readTransmission.Address        = slaveAdr;
readTransmission.AddressMode    = AddressMode7Bit;
readTransmission.BusSpeed       = I2C_BUS_SPEED_400KHZ;
readTransmission.DataLength     = sizeof(readBuf);
readTransmission.pBuffer        = readBuf;

status = DeviceIoControl(
                      fileHandler,
                      IOCTL_I2C_EXECUTE_READ,
                      NULL,
                      0,
                      &readTransmission,
                      sizeof(readTransmission),
                      NULL,
                      &Overlapped);

if(status || (GetLastError() == ERROR_IO_PENDING))
{
    status = GetOverlappedResult(
                      fileHandler,
                      &Overlapped,
                      &bytesReturned,
                      TRUE);
    if(status)
    {
        /****
        * Now readBuf contains data that read from slave device.
        ****/
    }
}
```

IOCTL_I2C_EXECUTE_WRITE Code Example

```
#include “public.h”

I2C_SINGLE_TRANSMISSION writeTransmission;
UCHAR writeBuf[BUF_SIZE] = {};
UINT16 slaveAdr         = 0x1C;


writeTransmission.Address        = slaveAdr;
writeTransmission.AddressMode    = AddressMode7Bit;
writeTransmission.BusSpeed       = I2C_BUS_SPEED_400KHZ;
writeTransmission.DataLength     = sizeof(writeBuf);
writeTransmission.pBuffer        = writeBuf;

status = DeviceIoControl(
                        fileHandler,
                        IOCTL_I2C_EXECUTE_WRITE,
                        &writeTransmission,
                        sizeof(writeTransmission),
                        NULL,
                        0,
                        NULL,
                        &Overlapped);

if(status || (GetLastError() == ERROR_IO_PENDING))
{
    status = GetOverlappedResult(
                        fileHandler,
                        &Overlapped,
                        &bytesReturned,TRUE);
    if(status)
    {
        /****
        * Now data in writeBuf have been transmitted to slave device.
        ****/
    }

}
```

IOCTL_I2C_EXECUTE_SEQUENCE Code Example

```
#include “public.h”

I2C_SEQUENCE_TRANSMISSION_ENTRY(2) sequence;
sequence.Size = 2;

USHORT regAddr = 0x0000;
UCHAR  readBuf[BUF_SIZE] = {};
UINT16 slaveAdr         = 0x1C;
UINT32 delayInUs        = 100;

/* Initialize write item in sequence*/
I2C_SEQUENCE_ITEM_INIT(
                        sequence.List[0],
                        AddressMode7Bit,
                        slaveAdr,
                        I2C_BUS_SPEED_400KHZ,
                        SpbTransferDirectionToDevice,
                        delayInUs,
                        sizeof(regAddr),
                        &regAddr);


/* Initialize read item in sequence*/
I2C_SEQUENCE_ITEM_INIT(
                       sequence.List[1],
                       AddressMode7Bit,
                       slaveAdr,
                       I2C_BUS_SPEED_400KHZ,
                       SpbTransferDirectionFromDevice,
                       delayInUs,
                       sizeof(readBuf),
                       readBuf);

status = DeviceIoControl(
                       fileHandler,
                       IOCTL_I2C_EXECUTE_SEQUENCE,
                       NULL,
                       0,
                       &sequence,
                       sizeof(sequence),
                       NULL,
                       &Overlapped);

if(status || (GetLastError() == ERROR_IO_PENDING))
{
    status = GetOverlappedResult(
```

```
                        fileHandler,
                        &Overlapped,
                        &bytesReturned,TRUE);
    if(status)
    {
        /****
        * Now data in regAddr have been transmitted to slave device,
        * and readBuf contains data read from slave device.
        * No STOP bit between item_0 and item_1.
        ****/
    }

}
```

3.5.3 Close Device

Once all operations related to the I2C driver are finished the device handle must free the application by calling the Win32 API CloseHandle.

```
CloseHandle(hHandle);
```

CHAPTER 4
Serial Peripheral Interface (SPI) Driver

This section provides the programming details of the Serial Peripheral Interface (SPI) driver for Windows. This includes information about the interfaces exposed by the driver and how to use those interfaces to drive the SPI hardware. The current implementation of the driver exposes the interfaces through Input/Output Controls (IOCTLs), which can be called from the application (user mode) using the Win32 API DeviceIoControl (refer to the MSDN documentation for more details on this API).

The SPI bus is a communication bus that operates in full duplex mode. Devices communicate in master/slave mode, in which the master device initiates the data transfer. The SPI hardware supports four different modes for communication.

4.1 Driver Features

The SPI Driver allows setting different configurations for SPI hardware. It supports:

- Master mode only
- Supports one SPI peripheral only
- Setting serial clock rate for transfer up to 15 Mbps
- Different modes – Mode 0, Mode 1, Mode 2 and Mode 3

4.2 Interface Details

The below Table 1 lists the IOCTLs supported by the SPI driver.

No	IOCTL	Description
3	IOCTL_SPI_READ	This IOCTL is used to read information from the devices connected to the SPI hardware.
4	IOCTL_SPI_WRITE	This IOCTL is used to write information to the devices connected to the SPI hardware.
5	IOCTL_SPI_SEQUENCE	This IOCTL is used to write sequence information to the devices connected to the SPI hardware.

4.3 IOCTL Usage Details

4.3.1 IOCTL_SPI_READ

This IOCTL is used to perform the read operation.

```
void Read(ULONG len)
{
    BOOL result;
    ULONG bytesRead;
    SPI_TRANSFER_BUFFER spiTransmissionData;

    UCHAR *ResponseBuffer = (UCHAR*)malloc(len);

    if(ResponseBuffer == NULL) return;

    spiTransmissionData.BitsPerEntry = 8;
    spiTransmissionData.ConnectionSpeed = 125000;
    spiTransmissionData.DataLength = len;
    spiTransmissionData.Mode = 0;
    spiTransmissionData.pBuffer = ResponseBuffer;

    DEBUG("WRITE SIZE %d", sizeof(spiTransmissionData));

    result = (BOOLEAN)DeviceIoControl(gSPIHdr,
                                      (DWORD)IOCTL_SPI_READ,
                                      nullptr,
                                      NULL,
                                      &spiTransmissionData,
                                      sizeof(spiTransmissionData),
                                      NULL,
                                      &gOverLapped);

    if(result || (GetLastError() == ERROR_IO_PENDING))
    {
        result = GetOverlappedResult(gSPIHdr, &gOverLapped, &bytesRead, TRUE);
        if(result)
        {
            DEBUG("IOCTL_SPBSPII2C_READ Succeed.\n");
            InfoBuffer(ResponseBuffer, bytesRead);
        }
    }
    else
    {
        ERRO("IOCTL_SPBSPII2C_READ Failed. ERRO Code: %x\n", GetLastError());
    }

    free(ResponseBuffer);
}
```

4.3.2 IOCTL_SPI_WRITE

This IOCTL is used to perform the read operation.

```
void Write(UCHAR* data, ULONG len,  UCHAR mode, ULONG speed)
{
    BOOL result;
    ULONG bytesWritten;
    SPI_TRANSFER_BUFFER spiTransmissionData;

    spiTransmissionData.BitsPerEntry = 8;
    spiTransmissionData.ConnectionSpeed = speed;
    spiTransmissionData.DataLength = len;
    spiTransmissionData.Mode = mode;
    spiTransmissionData.pBuffer = data;

    result = (BOOLEAN)DeviceIoControl(gSPIHdr,
                                      (DWORD)IOCTL_SPI_WRITE,
                                      &spiTransmissionData,
                                      sizeof(spiTransmissionData),
                                      nullptr,
                                      0,
                                      &bytesWritten,
                                      &gOverLapped);

    if(result || (GetLastError() == ERROR_IO_PENDING))
    {
        result = GetOverlappedResult(gSPIHdr, &gOverLapped, &bytesWritten, TRUE);
if(result)
        {
              DEBUG("IOCTL_SPBSPII2C_WRITE Succeed. Bytes Written %d\n", bytesWritten);
            DebugBuffer(data, len);
        }
    }
    else
    {
        ERRO("IOCTL_SPBSPII2C_WRITE Failed. ERRO Code: %x\n", GetLastError());
    }
}
```

4.3.3 IOCTL_SPI_SEQUENCE

This IOCTL is used to perform the sequence operation.

```
void Sequence(UCHAR command[], ULONG  commandLen, ULONG readSize, UCHAR mode, ULONG
speed)
{
    ULONG SequenceNum = 0;
    ULONG reqnum = 0;
    ULONG entryindex = 0;
    BOOL result;
    ULONG bytesReturned;
    UCHAR *responceBuffer;

    SPI_TRANSFER_LIST squence_buffer;
```

```
    squence_buffer.Size = 2;

    responceBuffer = (UCHAR*)malloc(readSize);

    if(responceBuffer == NULL) return;


    LPSS_SPI_SEQUENCE_BUFFER_ENTRY_ADD(squence_buffer, 0, commandLen,
       SpbTransferDirectionToDevice, 0, command);
    LPSS_SPI_SEQUENCE_BUFFER_ENTRY_SETCONFIG(squence_buffer, 0, mode, speed);

    LPSS_SPI_SEQUENCE_BUFFER_ENTRY_ADD(squence_buffer, 1, readSize,
       SpbTransferDirectionFromDevice, 0, responceBuffer);
    LPSS_SPI_SEQUENCE_BUFFER_ENTRY_SETCONFIG(squence_buffer, 1, mode, speed);

    result = (BOOLEAN)DeviceIoControl(gSPIHdr,
                                      (DWORD)IOCTL_SPI_SEQUENCE,
                                      &squence_buffer,
                                      sizeof(squence_buffer),
                                      nullptr,
                                      NULL,
                                      &bytesReturned,
                                      &gOverLapped);

    if(result || (GetLastError() == ERROR_IO_PENDING))
    {
        result = GetOverlappedResult(gSPIHdr, &gOverLapped, &bytesReturned, TRUE);
        if(result)
        {
            DEBUG("Read Succeed.Bytes: %d\n", bytesReturned);

            DebugBuffer(command, commandLen);

            if(readSize > 0)
            {
                InfoBuffer(responceBuffer, readSize);
            }
        }
        else
        {
            ERRO("Read Failed. ERRO Code: %x\n", GetLastError());
        }
    }
    else
    {
        ERRO("Read Failed. ERRO Code: %x\n", GetLastError());
    }

    free(responceBuffer);
}
```

4.4 Structures and Macros

4.4.1 Enumerations

Name	Description
IO_SPI_MODE_0	This specifies MODE 0, Clock Polarity =0 ,Clock Phase = 0
IO_SPI_MODE_1	This specifies MODE 1, Clock Polarity =0 ,Clock Phase = 1
IO_SPI_MODE_2	This specifies MODE 2, Clock Polarity =1 ,Clock Phase = 0
IO_SPI_MODE_3	This specifies MODE 3, Clock Polarity =1 ,Clock Phase = 1

4.4.2 SPI SEQUENCE STRUCT and MICROS

```
#define MAX_SEQUENCE_ENTRY_SIZE     2
typedef enum SPB_TRANSFER_DIRECTION
{
    SpbTransferDirectionNone,
    SpbTransferDirectionFromDevice,
    SpbTransferDirectionToDevice,
    SpbTransferDirectionMax
}
SPB_TRANSFER_DIRECTION, *PSPB_TRANSFER_DIRECTION;
__pragma(pack(push, 4))
typedef struct _SPI_TRANSFER_BUFFER
{
    UCHAR       Mode;
    UCHAR       BitsPerEntry;
    ULONG       ConnectionSpeed;

    UINT32      DataLength;
    PVOID       pBuffer;
} SPI_TRANSFER_BUFFER, *PSPI_TRANSFER_BUFFER;

typedef struct _SPI_TRANSFER_LIST_ENTRY
{

    SPB_TRANSFER_DIRECTION      Direction;
    UINT64                      DelayInUs;

    SPI_TRANSFER_BUFFER         SpiTransferBuffer;
} SPI_TRANSFER_LIST_ENTRY, *PSPI_TRANSFER_LIST_ENTRY;

typedef struct _SPI_TRANSFER_LIST
{
    UINT64                      Size;
    SPI_TRANSFER_LIST_ENTRY     List[MAX_SEQUENCE_ENTRY_SIZE];
} SPI_TRANSFER_LIST;

__pragma(pack(pop))

#define LPSS_SPI_SEQUENCE_BUFFER_ENTRY_ADD(SEQUENCE, _INDEX_, _LENGTH_, _DIRECTION_,
_DELAY_, _BUFFER_)       \
```

```
    (SEQUENCE).List[(_INDEX_)].SpiTransferBuffer.DataLength    = _LENGTH_  ;    \
    (SEQUENCE).List[(_INDEX_)].Direction     = _DIRECTION_  ;                  \
    (SEQUENCE).List[(_INDEX_)].DelayInUs     = _DELAY_  ;                       \
    (SEQUENCE).List[(_INDEX_)].SpiTransferBuffer.pBuffer       = _BUFFER_  ;

#define LPSS_SPI_SEQUENCE_BUFFER_ENTRY_SETCONFIG(SEQUENCE, _INDEX_, _MODE_,
_CONNECTIONSPEED_)       \
    (SEQUENCE).List[(_INDEX_)].SpiTransferBuffer.Mode                = _MODE_  ;\
    (SEQUENCE).List[(_INDEX_)].SpiTransferBuffer.ConnectionSpeed     =
_CONNECTIONSPEED_  ;
```

4.5 Error Handling

Since the IOCTL command is implemented using the Windows* API, the return value of the call is dependent on and defined by the OS. On Windows*, the return value is a non-zero value. If the error is detected within or outside the driver, an appropriate system defined value is returned by the driver.

4.6 Programming Guide

This section describes the basic procedure for using the SPI driver from a user mode application. All operations are through the IOCTLs exposed by the SPI driver. Refer to Section 4.3 for details on the IOCTLs. The steps involved in accessing the GPIO driver from the user mode application are described below:

- Open the device.
- Perform read/write operations.
- Close the device.

4.6.1 Open Device

SPI driver is opened using the Win32 CreateFile API. To retrieve the device name, see below explanation.

Using GUID Interface Exposed by the Driver

A device interface class is a way of exporting device and driver functionality to other system components, including other drivers, as well as user-mode applications. A driver can register a device interface class, and then enable an instance of the class for each device object to which user-mode I/O requests might be sent. The SPI driver registers the following interface.
This is defined in spi_public.h.

No	Interface Name
1	GUID_BYT_SPI_DEVICE_INTERFACE

Device interfaces are available to both kernel-mode components and user-mode applications. User-mode code can use SetupDiXxx functions to find out about registered, enabled device interfaces. Please refer the following site to get the details about SetupDiXxx functions.

http://msdn.microsoft.com/en-us/library/ff549791.aspx

4.6.2 Read and Write Operation

IOCTL_SPI_READ and IOCTL_SPI_WRITE are used for read and write operations respectively.

See Section 4.3.1 and 4.3.2 section.

4.6.3 Sequence Operation

IOCTL_SPI_SEQUENCE is used to perform sequence operation.

Following example shows how to read the EEPROM data by sequence. Before read data from EEPROM, should first write the EEPROM address to the SPI device. So this sequence will include a write operation and read operation.

refer to 4.3.3 section.

4.6.4 Close Device

Once all the operations related to the SPI driver are completed, the device handle must be freed by the application by calling the Win32 API CloseHandle.

```
CloseHandle(hHandle);
```